University of Twente

Internship Axini

Company Supervisor:
Dr. Ir. H.M. van der Bijl

Author:
Vincent Bloemen

University Supervisor:
Prof. Dr. J.C. van de Pol

Internship period:
February 1, 2014 - May 9, 2014

May 9, 2014

Preface and acknowledgements

From Februari 1st 2014 until May 9th 2014 I did my internship at the Dutch company Axini. The internship was carried out within the Methods and Tools for Verification (MTV) Master program at the University of Twente.

The internship assignment concerned the subject of constraint programming. Several extensions were implemented in the GNU Prolog tool, allowing it to be more suitable for practical applications. For this, a joint collaboration with the developer of GNU Prolog, Daniel Diaz, was realized. Near the end of the internship period, I was given the opportunity to write a paper about my work on GNU Prolog. This paper is currently (as of writing this report) in submission for the CICLOPS conference (Colloquium on Implementation of Constraint and LOgic Programming Systems). The contents of the paper are included in this report.

I would like to thank several people for helping me with the internship. I would like to thank my supervisor, Machiel van der Bijl, for making the internship possible and for his great guidance throughout the period. I would also like to thank Jaco van de Pol, my university supervisor, also for setting up the internship and for helping me to keep a close eye on the details during the project. Many thanks go to Daniel Diaz, for 'co-supervising' me and helping me quickly with problems I faced during the process of extending GNU Prolog. Furthermore, he brought up the CICLOPS conference and I thank him for his contribution in the paper. Salvador Abreu, also a developer of GNU Prolog, should be thanked as well, as he also helped me with writing the paper. Finally I would like to thank everyone at Axini, for all the shared knowledge and certainly not unimportant, for the great time I had during the internship.

Summary

In this project, several extensions to the GNU Prolog tool have been implemented. The underlying reason for these extensions is to improve the practical applicability of the tool for Axini's purposes. This is because Axini makes use of GNU Prolog's finite domain constraint solver in its Model-based Testing tool, called TestManager. The intended extensions for GNU Prolog constraint solver are:

- to include negative values in the finite domain variables.
- to improve the usability by preventing overflows from occurring.
- to reduce/remove the limitations of the sparse domain representation.

This report discusses the methodology and techniques used for implementing the extensions. A focus is laid on correctness and performance. The report analyzes each extension in detail, after which possible alternatives are discussed. The chosen approaches are motivated and the implementation is discussed with the use of examples.

The results show that each extension is successfully implemented. The performance evaluation of this initial implementation shows encouraging results, with a limited slowdown factor compared to the original constraint solver. The developers of GNU Prolog are pleased with the results and aim to incorporate the extensions in a future release of GNU Prolog.

Contents

1 Introduction

Axini is a company that specializes in developing tools for creating high-quality software. Axini's product, called *TestManager*, tests systems thoroughly while remaining quick. For this, *Model-Based Testing* is used. In Model-Based Testing, test cases are automatically derived from a model of the system. Thereafter, these tests are executed on the system and the results are analyzed. System models are described as automatons (transition systems) of which the input- and output channels are used for describing the control flow of the system. Axini uses *symbolic transition systems* for these models, making TestManager highly suitable for operating with parallel systems and data flows. Symbolic transition systems describe transitions using guards: logical formulas over a number of variables. These formulas are evaluated with *constraint solving* to deduce concrete tests. For constraint solving, GNU Prolog[4] is used; a free Prolog compiler with the extension of constraint solving on finite domains.

As a simplified example, consider the following pseudo-code *if*-statement (with a preceding input request for the integer variable X).

```
input(X);
if (X > 1 && X < 10) {
  // do action
}
```

This simple program constrains the integer variable X to only take values in the range `[2..9]` for 'action' to be executed. The TestManager will search for an assignment for X that evaluates the constraint to *true*. The GNU Prolog constraint solver is used for this purpose, as demonstrated below.

```
| ?- X #> 1, X #< 10, fd_labeling(X,[value_method(max)]).
X = 9 ?
yes
```

Here, a finite domain variable for X is constructed and its domain is reduced after applying the constraints `X > 1` and `X < 10`. The operators `#>` and `#<` are used for applying the respective mathematical operators $>$ and $<$ on finite domain variables. For example, X can initially take all values in the domain `[0..max_int]`, after applying the constraint `X #< 10` the domain for X is updated to `[0..9]`. Eventually, X represents the domain `[2..9]`. The predicate `fd_labeling` is used to obtain a value v such that $v \in X$ (and $\forall u \in X : v \geq u$). This value, 9, is then assigned to X in the generation of a test case.

For the internship, close attention is paid to the constraint solver. As Section 3 will discuss, several limitations exist in the current version of the GNU Prolog solver. The internship focuses on addressing these limitations and proposing solutions to reduce these to a minimum.

For the remainder of the report, Section 2 contains background information on constraint solving in general and the current implementation of the GNU Prolog solver. Section 3 discusses the limitations of the current solver and formulates these into concrete research questions. Sections 4,5 and 6 discuss the design choices and methodology used for answering the research questions. An overview of the modifications in the solver and comparative results with respect to the original version can be found in Section 7. Section 8 provides some directions for further improving the overall performance. A conclusion is provided in Section 9. Finally, the appendices provide information regarding the structure and development of the GNU Prolog constraint solver.

2 Background information

Constraint Programming [1, 3, 9] emerged, in the late 1980s, as a successful paradigm to tackle complex combinatorial problems in a declarative manner [5]. Initially, the internal designs of constraint solvers were only accessible by a few highly specialized engineers.

One major advancement in the development of constraint solvers over Finite Domains (FD) is the article from Van Hentenryck et al. [6]. This paper proposed a "glass-box" approach based on a single *primitive constraint* whose understanding is immediate. This primitive takes the form X `in` r, where X is an FD variable and r denotes a *range* (i.e. a set of values). An X `in` r constraint enforces X to belong to the range denoted by r *which can involve other FD variables.* An X `in` r constraint which depends on another variable Y becomes *store sensitive* and must be (re)activated each time Y is updated, to ensure the consistency.

It possible to define high-level constraints, such as equations or inequations, in terms of X `in` r primitive constraints. It is worth noticing that these constraints are therefore not built into the theory. From the theoretical point of view, it is only necessary to work at the primitive level as there is no need to give special treatment to high-level constraints. This approach yielded significant advances in solvers.

2.1 The GNU Prolog FD Solver

The GNU Prolog solver follows the "glass-box" approach introduced by Van Hentenryck et al. [6], in which the authors propose the use of a single *primitive constraint* of the form X `in` r, where X is an FD variable and r denotes a *range* (i.e. a set of values). An X `in` r constraint enforces X to belong to the range denoted by r which can be constant (e.g. the interval `[1..10]`) but can also use the following *indexicals*:

- `dom(`Y`)` representing the whole current domain of Y.
- `min(`Y`)` representing the minimum value of the current domain of Y.
- `max(`Y`)` representing the maximum value of the current domain of Y.
- `val(`Y`)` representing the final value of the variable of Y (when its domain is reduced to a singleton). A constraint using this indexical is postponed until Y is instantiated.

An X `in` r constraint which uses an indexical on another variable Y becomes *store sensitive* and must be (re)activated each time Y is updated to ensure the consistency. Thanks to X `in` r primitive constraints it possible to define high-level constraints such as equations or inequations. All solvers offer a wide variety of predefined (high-level) constraints to the programmer. Nevertheless, the experienced user can define his own constraints if needed.

The current FD solver of GNU Prolog is also based on indexicals. Its implementation is widely based on its predecessor, `clp(FD)` [2]. The rest of this section explains some aspects of the current implementation which are important later on. The interested reader can refer to [4] for missing details.

2.1.1 The FD definition language

The original X `in` r is not expressive enough to define all needed constraints in practice. For this, the FD language is designed: a specific language to define the constraints of the GNU Prolog solver. Figure 1 shows the definition of the constraint $A \times X = Y$ in the FD language:

```
ax_eq_y(int A, fdv X, fdv Y)                 /* here A != 0 */
{
  start X in min(Y) /> A .. max(Y) /< A       /* X = Y / A */
  start Y in min(X) *  A .. max(X) *  A       /* Y = X * A */
}
```

Figure 1: Definition of the constraint $A \times X = Y$ in the FD language.

The first line defines the constraint name (`ax_eq_y`) and its arguments together with their types (`A` is expected to be an integer, `X` and `Y` FD variables). The `start` instruction installs and activates an X `in` r primitive. The first primitive computes X from Y in the following way: each time a bound of Y is modified the primitive is triggered to reduce the domain of X accordingly. The operator `/>` (resp. `/<`) denote division rounded upwards (resp. downwards). Similarly, the second primitive updates (the bounds) of Y with repect to X. This is called *bound consistency* [1]: if a *hole* appears inside the domain of X (i.e. a value V, with `min(X)` $< V <$ `max(X)`, has beed removed from the domain of X), the corresponding value $A \times V$ will not be removed from the domain of Y. If wanted, such a propagation (called *domain consistency*) could be specified using the `dom` indexical.

As an example, presuming that the FD variable X represents the domain `[1..4]`, the bound consistent constraint `X * 2 #= Y` updates the domain for Y to `[2..8]` while the domain consistent constraint `X * 2 #=# Y` updates the domain for Y to `[2:4:6:8]`.

A compiler (called `fd2c`) translates an FD file to a C source file. The use of the C language as target is motivated by the fact that the GNU Prolog system is written in C (so the integration is simple) but mainly by the fact that modern C compilers produce very optimized code (this is of prime importance considering that a primitive constraint can be awoken several thousand times in a resolution). When compiled such a definition gives rise to different C functions:

- the *main function*: a public function (`ax_eq_y`) which mainly creates an environment composed of the 3 arguments (A, X, Y) and invokes the installation functions for the involved X `in` r primitives.
- the *installation function*: a private function for each X `in` r primitive which is responsible for the installation of the primitive. This consists of installing the dependencies (e.g. add a new dependency to Y, so that each time Y is modified the primitive is re-executed to update X) and the execution function is invoked (this is the first execution of the primitive).
- the *execution function*: a private function for each X `in` r primitive which computes the actual value of r and enforces $X \subseteq r$. This function will be (re)executed each time an FD variable appearing in the definition of r is updated.

2.1.2 Internal domain representations

There are two main representations of a domain (range):

- *MinMax*: only the *min* and the *max* are stored. This representation is used for intervals (e.g. `[0..max_int]`) and will also be referred to as the *interval representation.*
- *Sparse*: this representation is used as soon as a hole appears in the domain of the variable. In that case, in addition to the *min* and the *max*, a bit-vector is used to record each value of the range.

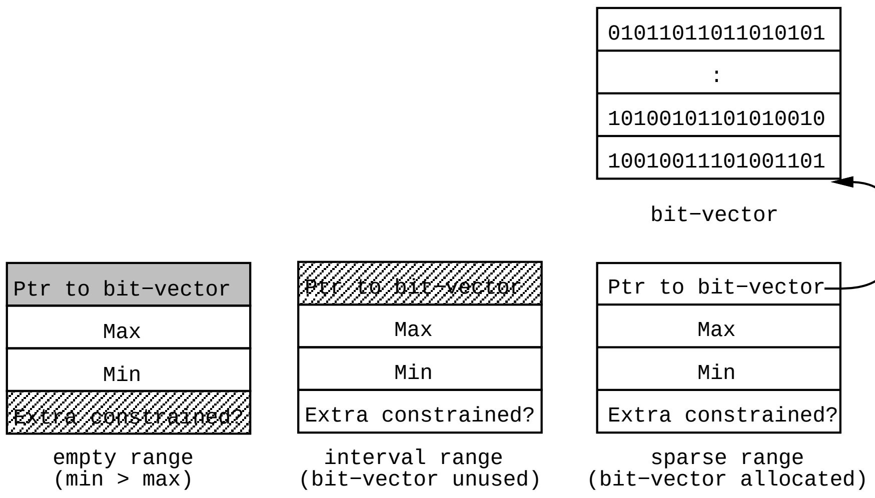


Figure 2: Representations of a range.

When an FD variable is created it uses a *MinMax* representation, where $min = 0$ and max[1]$= 2^{28} - 1$ initially. As soon as a "hole" appears it transparently switches to a *Sparse* representation which uses a bit-vector. For efficiency reasons all bit-vectors can exclusively store values in the range `[0..fd_vector_max]`. By default `fd_vector_max` equals 127 and can be redefined via an environment variable or via a built-in predicate (this should be done before any constraint is told). When a range becomes *Sparse*, some values

[1]GNU Prolog uses a 32 bit Prolog is used internally, which uses 3 bits to tag the data due to dynamic typing.

are possibly lost if `fd_vector_max` is less than the current *max* of the variable. To inform the user of this source of incompleteness, GNU Prolog maintains a flag to indicate that a range has been *extra constrained* by the solver (via an imaginary constraint *X in* `[0..fd_vector_max]`). The flag *extra_cstr* is associated to each range and is updated by all operations, e.g. the intersection of two ranges is extra-constrained iff both ranges are extra constrained, thus the resulting flag is the logical *and* between the two flags. If a failure occurs on a variable whose domain is extra constrained a message is displayed to inform the user that some solutions can be lost since bit-vectors is too small. Finally, an empty range is represented with $min > max$. This makes it possible to perform an intersection between R_1 and R_2 in *MinMax* mode simply with $[Max(min(R_1), min(R_2))\,..\,Min(max(R_1), max(R_2))]$ which returns $min > max$ if either R_1 or R_2 is empty. Figure 2 shows the different representations of a range.

3 Problem description

For the internship assignment, an extension to the GNU Prolog constraint solver is desired. While this solver in its current state is very efficient and already includes a wide variety of predefined predicates, it is limited in other areas. These limitations of the current solver cause its applicability for Axini's purposes to be cumbersome. For the scope of internship assignment, it is desired to reduce the limitations of the current solver as much as possible.

To formulate the assignment concretely, three research questions are constructed, of which each concerns a limitation that should be resolved. The following parts briefly discuss the limitations in the current GNU Prolog version.

3.1 Supporting negative values

The current solver is only able to use positive values in the FD variables, meaning that an FD variable's domain is always a subset of the range `[0..max_int]`. In the symbolic transition systems from Axini, negative values are possible in the data flow. Currently a work-around is used for dealing with negative numbers: each value is incremented with a constant (to make negative values positive) after which the solver can be used on the new value, finally the resulting domain of the solver is decremented with the constant to retrieve the actual value. Although this technique works for simple constraints, for more complex operations (such as multiplications) the results quickly become incorrect. Even though it is theoretically possible to "translate" each problem to natural numbers, there are several drawbacks: the translation is error-prone, the resulting programs are difficult to read, and the performance can be significantly affected. For both Axini and advancements of GNU Prolog it is beneficial to structurally incorporate negative values in the FD domains. The corresponding research question is as follows:

> **Research question 1:** How can GNU Prolog be extended to support negative values in the finite domain solver?

3.2 Overflow prevention

Due to efficiency reasons, GNU Prolog does not check for overflows. This means that without preliminary domain definitions (i.e. reducing the initial domain for FD variables to a smaller domain) the constraint `X * Y #= Z` will fail due to an overflow when computing the upper bound for Z: the maximum values that both X and Y can take is `max_int`, resulting in the computation `max_int`×`max_int` which translates to $(2^{28} - 1) \times (2^{28} - 1)$. This computation will overflow 32 bit and causes the upper bound of Z to be negative, the constraint will then fail (because `max(Z)` > `min(Z)`). For the user it is not always possible to reduce the initial domains for the FD variables and to prevent overflows from occurring. This is also the case for Axini: it might be possible to choose minimum and maximum values for variables in a particular guard, but since data continues to flow further through the model, the initially chosen bounds may not be sufficient. More generally, the lack of overflow prevention puts a (cumbersome) constraint on the user. To improve the usability of GNU Prolog, the following research question is formed:

> **Research question 2:** What measures can be taken to prevent overflows from occurring in the GNU Prolog finite domain solver?

3.3 Redesigning the sparse representation

As already mentioned in Section 2.1.2, when the internal domain representation of an FD variable switches to the *Sparse* representation, some values could be lost. For example in the following case. Presume that the FD variable X represents the domain `[10..300]`. Now suppose that the constraint `X #\= 50` is applied (meaning that X cannot take the value 50). Because the maximum value[2] that the *Sparse* representation can take is 127, the new domain for X is `[10..49:51..127]`. The values `[128..300]` are now lost (and the *extra_cstr* flag is set). Obviously this is a (severe) constraint on sparse FD variables in many practical purposes. For Axini, and GNU Prolog in general, loss of variables should be prevented as much as possible by redesigning the *Sparse* representation. The corresponding research question is as follows:

> **Research question 3:** How can GNU Prolog be modified to prevent variable loss in the *Sparse* representation for FD variables?

The following three sections discuss the process of 'answering' each research question. Note that the three research questions overlap to some extent (e.g. a multiplication with two negative numbers can also cause an overflow). Clearly, these newly introduced problems are also addressed. Two important aspects to consider in each question are *correctness* and *performance*. Correctness is determined by means of extensive testing and comprehensive manual verification; each of the three sections contains a part on how the verification process is conducted. The performance of the modified solver is compared with the original version using a number of benchmark tests.

4 Supporting negative values

This section describes how the inclusion of negative values in FD variables is realized. First it is shown why the current implementation does not support negative values. Then the problem is addressed by mainly focusing on the implementation. Finally the process of analyzing and verifying the correctness of the adaptations is mentioned. The proposed modifications mainly concern updates in the mathematical predicates (described in the FD language), more information on the underlying control flow structure can be found in Appendix A.

4.1 Current limitations

The current implementation does not support negative values, FD variables stay within the bounds `[0..max_int]`. Adding support for negative values seems obvious at a first glance, however some attention has to be paid. The modifications concern constraints whose current implementation implicitly utilize the fact that values are always positive, which is no longer valid. Other modifications concern constraints which are sign sensitive from the interval arithmetical point of view. This is the case for multiplication: if X is in $[A..B]$ then $-X$ is in $[-B..-A]$. The constraint $A \times X = Y$, whose current definition is presented in Figure 1, is reconsidered. Presuming that A can be negative, the current definition will not update the domains of X and Y correctly: in that case X will be constrained to $[\lceil \frac{min(Y)}{A} \rceil .. \lfloor \frac{max(Y)}{A} \rfloor]$ which produces an empty interval since $min(X) > max(X)$. To support negative values in FD variables, this instance, as well as other arithmetical constraints require updating to handle negative values properly.

4.2 Method and approach

One possible approach to deal with negative numbers is to construct a mapping for negative values to natural numbers so that the arithmetic constraints can continue to operate strictly on the positive domain. Another approach is to update the constraints to be fully functional for both positive and negative domains. The former is undesirable since the translation quickly becomes cumbersome and would carry a considerable performance impact. Aside from that, several operations such as taking the power or root are affected by the variable sign. As the latter approach is less error-prone and more robust, this approach is chosen and thus several arithmetic constraints need to be reformulated.

[2]The upper bound can be incremented by the user but this is not desired, as discussed in Section 6.

First, the initial domain bounds of FD variables are updated to range in `[min_int..max_int]` where `min_int` $= -2^{28} + 1 =$ `-max_int`[3]. To remain backwards compatible, an environment variable is created that, if set, will use the original bounds for FD variables (i.e. setting `min_int` $= 0$).

On updating the arithmetic constraints, all possible cases for each FD variable need to be considered, that is < 0, $= 0$ and > 0 for both the *min* and *max* of the variable (see Section 4.3 for more information). For instance, the $A \times X = Y$ constraint from Figure 1 is updated as follows:

```
ax_eq_y(int A, fdv X, fdv Y)               /* A != 0 */
{
 start X in ite(A>0, min(Y), max(Y)) /> A    /* X = Y / A */
         .. ite(A>0, max(Y), min(Y)) /< A
 start Y in ite(A>0, min(X), max(X)) * A     /* Y = X * A */
         .. ite(A>0, max(X), min(X)) * A
}
```

where `ite` represents an if-then-else expression (corresponding to the C operator `?:`). This modification ensures that for all interpretations of A, X and Y the domains are updated correctly as shown later.

A more complex example is the constraint $X^A = Y$, where X and Y are FD variables and A is an integer > 2. In the current version, this constraint is given as follows:

```
x_power_a_eq_y(fdv X, int A, fdv Y)          /* A > 2 */
{
 start Y in Power(min(X), A)..Power(max(X), A)
 start X in Nth_Root_Up(min(Y), A)..Nth_Root_Dn(max(Y), A)
}
```

With the introduction of negative values, the constraint is specified as:

```
x_power_a_eq_y(fdv X, int A, fdv Y)          /* A > 2 */
{
 start X in ite(is_even(A),
               min_root(min(X), min(Y), max(Y), A),
               ite(min(Y) < 0,
                   -Nth_Root_Dn(-min(Y), A),
                   Nth_Root_Up(min(Y), A)))
         .. ite(is_even(A),
               max_root(max(X), min(Y), max(Y), A),
               ite(max(Y) < 0,
                   -Nth_Root_Up(-max(Y), A),
                   Nth_Root_Dn(max(Y), A)))

 start Y in ite(min(X) < 0 && is_odd(A),
               Power(min(X), A),
               Power(closest_to_zero(min(X), max(X)), A))
         .. ite(min(X) < 0 && is_even(A),
               Power(Max(abs(min(X)), max(X)), A),
               Power(max(X), A))
}
```

here, a couple of C methods and macros are introduced:

- `Min` and `Max` are used to compute the minimum resp. maximum of two values.
- `is_even` and `is_odd` return whether the variable is even or odd.
- `min_root` and `max_root` calculate the minimum and maximum value of $\pm \sqrt[A]{Y}$ that lie in the bounds of `min(X)..max(X)`.

[3]Because 29 bits are used for representing integers, the most negative value is actually -2^{28}. However, `min_int` $= -2^{28}+1$ is chosen to keep consistent behavior with the positive domain, for instance when calculating $-X$.

- `Power` and `Nth_Root` refer to C methods that calculate the n^{th} power and n^{th} root of a variable.
- `closest_to_zero(A,B)` returns the closest value to 0 in the interval `[A..B]`.

In this specification, Y can only include negative values if X contains negative values and A is an odd number (e.g. $-2^3 = -8$). Similarly, if Y is strictly positive, X can only take negative values if A is an even number (e.g. $-2^4 = 16$). In short, the above constraint needs to distinguish between even and odd powers of X, which was originally unnecessary. With this definition, the following query correctly reduces the domains of X and Y:

```
|?- fd_domain([X,Y],-50,150), X ** 3 #= Y.
X = _#0(-3..5)
Y = _#17(-27..125)
```

The support for negative values in FD variables is achieved by carefully redesigning the arithmetic constraints. An obvious side-effect of the modifications is that some overhead is introduced, even when considering strictly positive FD variables. The benchmark tests, see Section 7.1 for the results, will show the impact of the modifications compared to the original solver.

4.3 Analysis and verification

Maintaining correctness of the FD solver is important, therefore each predicate is analyzed in detail. In multiplicative constraints, the signs of the variables are important to consider and for power operations it is also important to know whether a particular value is even or odd. For this reason, case distinctive measures are applied: all combinations of 'special' cases are manually evaluated. For the constraint $A \times X = Y$, the different cases to consider are < 0, $= 0$ and > 0 because of the multiplication. The evaluations for this constraint are represented in Table 1.

$min(X)$	$max(X)$	A	$min(Y)$	$max(Y)$
< 0	< 0	< 0	$max(X) \times A$	$min(X) \times A$
< 0	< 0	> 0	$min(X) \times A$	$max(X) \times A$
< 0	$= 0$	< 0	0	$min(X) \times A$
< 0	$= 0$	> 0	$min(X) \times A$	0
< 0	> 0	< 0	$max(X) \times A$	$min(X) \times A$
< 0	> 0	> 0	$min(X) \times A$	$max(X) \times A$
$= 0$	$= 0$	< 0	0	0
$= 0$	$= 0$	> 0	0	0
$= 0$	> 0	< 0	$max(X) \times A$	0
$= 0$	> 0	> 0	0	$max(X) \times A$
> 0	> 0	< 0	$max(X) \times A$	$min(X) \times A$
> 0	> 0	> 0	$min(X) \times A$	$max(X) \times A$

Table 1: Applying case distinctive measures for calculating the bounds of Y in the constraint $A \times X = Y$.

Because these results represent all possible distinct cases, it can be carefully assumed that this is a *complete* evaluation. Note that this is a simple predicate, more complex predicates are much more prone to human error, but the same algorithm is applied.

From Table 1, the resulting `X in r` constraints for Y can be derived. Initially every case can be implemented as a *switch*-structure with the resulting *min* and *max* bounds for Y. By analyzing the results in more detail it can be derived that only the sign of A influences the bound choices for Y. This results in the implementation that is presented in Section 4.2.

Besides this mathematical view of the constraints, two test suites, written in Prolog, are created that test a number of concrete constraints. The aim for these test suites is to verify whether the implementation is correct. More information about these test suites can be found in Appendix B.1.

An expected consequence of the inclusion of negative values, with the corresponding updates to the predicates, is that the performance of the FD solver is affected. A number of benchmark tests are executed using both the original and the updated version of GNU Prolog. The results of these tests and their performance differences are analyzed in Section 7.1.

5 Overflow prevention

5.1 Current limitations

The current implementation of GNU Prolog does not check for overflows. This means that without preliminary domain definitions for X, Y and Z, the non-linear constraint $X \times Y = Z$ will fail due to an overflow when computing the upper bound of the domain of $Z : (2^{28} - 1) \times (2^{28} - 1)$. In 32-bit arithmetic, this overflow causes a negative result for the upper bound and the constraint then fails since $min(X) > max(X)$.

At present, the user needs to adapt the variable bounds beforehand to prevent this constraint from failing. To reduce the burden to the user and improve the robustness of the solver, a better way of handling overflows is proposed.

5.2 Method and approach

There are two approaches to handle overflows. One is to report the problem via an ISO exception (e.g. `evaluation_error`), thereby informing the user that the domain definitions for the FD variables are too mild and should be made more restrictive. The other approach is to instrument the solver to detect overflows and cap the result. As placing less restrictions on the user and more robustness for the solver is desirable, the second approach is chosen.

The key idea behind preventing overflows is to detect when one would occur and provide means to restrict this from happening. For the solver this means that when a multiplication or power operation is applied in a constraint, an overflow preventive check should be considered. This can also be the case for other arithmetic operations.

Consider again the constraint $X \times Y = Z$. Because both $1 \times (2^{28}-1) = 2^{28}-1$ and $(2^{28}-1) \times 1 = 2^{28}-1$, the maximum value that both X and Y can take is $2^{28} - 1$. Therefore the following (and original implementation) for finding the domain for Z causes an overflow:

```
start Z in min(X) * min(Y) .. max(X) * max(Y)
```

For this case and similar instances, the following function is designed to cap results of arithmetic, thereby preventing overflows:

```
static int inline mult(int a, int b)
{
  int64_t res = ((int64_t) a) * ((int64_t) b);
  if (res > max_integer)
    res = max_integer;
  else if (res < min_integer)
    res = min_integer;
  return (int) res;
}
```

Since integers only need 29-bits, the 64-bit result is enough to check if an overflow occurs and cap the result if needed. In the constraint definitions, the standard multiplication gets replaced with a `mult` call when it could cause an overflow. For the $X \times Y = Z$ constraint, this is as follows:[4]

```
start Z in mult(min(X), min(Y)) .. mult(max(X), max(Y))
```

As a consequence, the $X \times Y = Z$ constraint now gives the following result:

```
| ?- X * Y #= Z.
X = _#3(-268435455..268435455)
Y = _#20(-268435455..268435455)
Z = _#37(-268435455..268435455)
```

[4]The constraint is further modified for negative values, along the same lines.

where `-268435455 = min_int` $= -2^{28} + 1 = -$`max_int`.

At first, `mult` was used for every applied multiplication in the constraint definitions. However, in some cases it is not necessary to check for overflows. For instance, consider the implementations for `ax_eq_y` and `x_power_a_eq_y` of Section 4.2. By first restricting the domain of X (in both cases), no overflow can occur when the domain of Y is calculated. However, in case the domain of Y is computed first, an overflow could happen. Note that such an optimization is not possible for some constraints, for instance in $X \times Y = Z$, since the domains of X and Y do not necessarily get reduced.

While some predicates required updating, the arguments for these predicates could also suffer from overflow errors. This is because the C code is able to handle values that exceed 29 bit integers, but the translation back to Prolog integers can be erroneous. If the value 2^{28} is stored in a 29 bit integer, due to the two's complement representation of a value, the Prolog value would be falsely set to -2^{28}. As a solution for this problem, a preventive check takes place before an integer value is translated to its Prolog representation. If this value is smaller than `min_int` or larger than `max_int`, the constraint fails. As an example, the folowing constraint will now correctly fail.

```
| ?- X #= 268435455 + 1.
no
```

In conclusion, even if several overflow problems could be resolved by rearranging the order of execution, in general it is necessary to take preventive measures.

5.3 Analysis and verification

For evaluating the correctness of the overflow preventive measures, multiple test cases were designed. These test cases were designed to test values at `min_int` (and `max_int`) and values just below and above this limit.

Furthermore, GNU Prolog supports both 32 and 64 bit machines (a 64 bit machine is used for development). While the underlying Prolog engine remains using 29 bits for integers, the 32 bit C code may also be affected by overflows. A testing environment for a 32 bit version is set up and the implementation is equivalently tested in this environment.

Similar to the inclusion of negative values, the implemented preventive measures create an overhead on the FD solver which should affect the performance. A number of benchmark tests are executed using both the original and the updated version of GNU Prolog. The results of these tests and their performance differences are analyzed in Section 7.1.

6 Redesigning the sparse representation

6.1 Current limitations

In the current implementation, when a *hole* appears in a domain, its representation is switched to the *Sparse* form, which stores domains using a static-sized bit-vector. The problem with this approach is that values which lie outside the range `[0..fd_vector_max]` are lost. An internal flag *extra_cstr* is set when this occurs to inform the user of lost values. Even though the user is able to globally set `fd_vector_max`, there are several problems with this representation:

- The user has to know the variable bounds in advance; an over-estimate of the domain size results in a waste of memory (and loss of efficiency).
- There is an upper-limit for `fd_vector_max` which is directly related to the available memory space in bits. Also note that doing operations on a large bit-vector can severely impact the performance.
- The current *Sparse* representation is unable to store negative values.

6.2 Method and approach

To deal with the limitations, a redesign is needed for the *Sparse* representation. Some research has been done in representing sparse domains [7, 8]. Considering the demands (remain efficient while taking away the limitations), there are several options for the redesign:

1. Use a list of *MinMax* chunks: store only the minimum and maximum values of consecutively set values. The values between two chunks are defined to be all unset.
2. Use a list of bit-vector chunks: use a bit-vector together with an offset to store all (un)set actual values. The values between two chunks can either be defined as all set or all unset (possibly defined per chunk with a variable).
3. A combination of (1) and (2): determine per chunk whether it should be a *MinMax* chunk or bit-vector chunk, so that the number of total chunks is minimal.

Other alternatives and variations on these designs are considered as well. Note that all suggested redesigns take away the limitations of the current design. Every option has its strengths and weaknesses compared to the others:

List of *MinMax* chunks:

+ Effective if the number of holes is small or large gaps exist in the domain.
– Loses efficiency (in both memory and performance) on a small domain with many holes.

List of bit-vector chunks:

+ Especially effective for small domains with many holes (due to bitwise operations).
– Loses efficiency on domains with large intervals and gaps.

A combination of the two:

+ Takes advantage of the strengths of both individual options.
– Extra overhead introduced for determining which representation to choose (and operations between *MinMax* chunks and bit-vector chunks need to be defined).

Le Clément et al. [7] provides a more in-depth analysis on the different representations with respect to their time complexities. Note that differences arise with specific operations on domains. For instance, a value removal is done more efficiently in a bit-vector while iteration is more efficient on *MinMax* chunks.

The initial choice is for the combination of the *MinMax* and bit-vector chunks because the extra overhead is presumed to not be a significant factor. As a start, a list of *MinMax* chunks was implemented. Its performance compared to the original *Sparse* implementation shows a limited slowdown factor, as discussed in Section 7.1. Because of these results (a slowdown is expected due to the new possibilities), the addition of a bit-vector is deemed unnecessary and is removed. The new implementation using a list of *MinMax* chunks will be discussed.

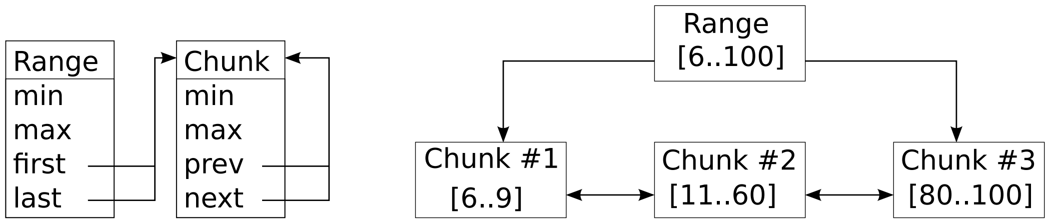


Figure 3: Left: UML diagram of the new *Sparse* range, right: example for representing the set of values {`6..9,11..60,80..100`}.

The range will initially use the *MinMax* representation, and it switches to the new *Sparse* representation by creating a *MinMax* chunk. It will then keep track of the first and last chunk of the list. The range maintains the absolute minimum and maximum of the chunk list (`range.min = range.first.min`), because these properties are often requested. The list is a doubly-linked list for efficient insertion and removal of chunks, each chunk maintains a minimum and maximum value. This representation is depicted in Figure 3.

For every two consecutive chunks c_1 and c_2, $c_1.max + 1 < c_2.min$; chunks are sorted and always have at least one unset variable between them. Furthermore, $chunk.min \leq chunk.max$.

Applying constraints on *Sparse* ranges is done by efficiently iterating over the chunks and updating these in place if possible. An example of this is provided in Table 2 for intersecting two *Sparse* ranges. The implementation only considers one chunk of each range at a time.

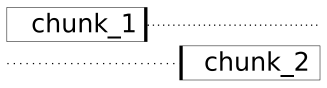
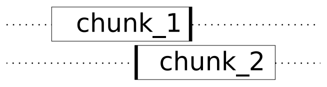
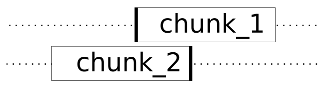
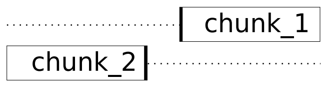
Depiction of each Case:	Action (in pseudo code):
	Remove `chunk_1` `chunk_1` = `chunk_1.next`
	`chunk_1.min` = Max(`chunk_1.min, chunk_2.min`) `chunk_1.max` = Min(`chunk_1.max, chunk_2.max`) `chunk_1` = `chunk_1.next`
	`chunk_1.min` = Max(`chunk_1.min, chunk_2.min`) `chunk_1.max` = Min(`chunk_1.max, chunk_2.max`) `chunk_2` = `chunk_2.next`
	`chunk_2` = `chunk_2.next`

Table 2: Implementation of the range intersect operation.

In case the solver backtracks, domains need to be stored and restored from the stack. Modifications on domains can cause its chunks to disperse in the memory, therefore all chunks of the domain are copied on top of the stack at backtrack-points. This prevents the solver from overwriting parts of the domain.

With this new implementation for the *Sparse* domain, it is now possible to store negative values and the domain bounds are not decreased anymore, thereby removing the use for the *extra_cstr* flag.

6.3 Analysis and verification

During the implementation process of the new *Sparse* representation, a large amount of methods required a redesign. A unit test suite is designed specifically for verifying the correctness of these methods. To give an example, a domain consistent multiplication of the domain `[-5..5]` with the value 2 is tested as follows:

```
r = create_range("-5..5");
Pl_Range_Mul_Value(r,2);
range_test("(-5..5)*2", r, "-10:-8:-6:-4:-2:0:2:4:6:8:10");
```

Appendix B.1 provides more detail on this unit test suite.

Originally, storage of a *Sparse* range only requires the location of the bit-vector te be known, which always uses the same space. With the new representation, it is impossible to know how many chunks a range will eventually have and where they are all stored. Because of this, extra attention is paid to the storage of chunks. For a number of test cases, the pointer addresses were carefully analyzed to make sure that no chunks are illegally overwritten.

As with the other extensions to the FD solver, the performance is affected by the redesign of the sparse domain. Besides comparing the performance differences of the original and the new version of GNU Prolog, special attention is paid on the origin for the differences. Using a tool called `callgrind`, the time spent in each method can be analyzed for specific test instances. Wherever possible, bottleneck methods are optimized to improve the overal performance. The results, discussed in Section 7.1, will furthermore explain the reasons for performance losses and gains in several test cases. This is a result of comparing the `callgrind` evaluations for the original and new version of GNU Prolog.

7 Results

This section presents an overview of the implemented features and an analysis on the performance impact when comparing the new version with the original implementation of GNU Prolog.

7.1 Performance analysis

The original FD constraint solver is compared to a version that includes the new extensions. Table 3 presents the total execution times (in milliseconds) for runs of several benchmarks. *Neg + Ovfl* consists of the negative values extension and the overflow prevention (the *Ovfl* extension is implemented simultaneously with *Neg*). *Neg + Ovfl + Dom* includes all three extensions. Times are measured on a 64-bit i7 Processor, 2.40GHz×8 with 8GB memory running Linux (Ubuntu 13.10).[5]

Program	Original	Neg + Ovfl		Neg + Ovfl + Dom	
	Time	Time	Speedup	Time	Speedup
`queens 29`	422	412	1.02	752	0.56
`queens ff 100`	154	155	0.99	229	0.67
`qg5 11` (×10)	626	597	1.05	897	0.70
`eq20` (×100)	185	244	0.76	257	0.72
`digit8 ff` (×100)	639	1,001	0.64	855	0.75
`magsq 11`	819	822	1.00	1,093	0.75
`langford 32`	551	550	1.00	708	0.78
`donald` (×10)	147	118	1.25	155	0.95
`multipl` (×10)	579	580	1.00	604	0.96
`partit 600`	247	208	1.19	251	0.98
`alpha` (×10)	410	394	1.04	414	0.99
`magic 200`	209	200	1.04	210	1.00
`crypta` (×1000)	763	641	1.19	621	1.23
`interval 256` (×10)	212	225	0.94	173	1.23
Average	426	439	**0.97**	517	**0.82**

Table 3: Performance impact of the extensions (times in ms).

The original implementation and the benchmark tests are solely designed for the positive domain. Therefore the domain bounds are restricted to positive values (using the environment variable discussed in Section 4.2), while making use of the updated constraint definitions. Multiple test runs show an estimated standard deviation of 3 milliseconds. The annotation (×10) indicates that the test time is formed from 10 consecutive executions (to reduce the effect of the deviation).

On average, the introduction of negative domains + overflow detection penalizes the benchmarks by 3%. This slowdown is an expected consequence of the increased complexity. The worst case is for `digit8 ff` with a 36% performance loss. The reason for this is because the square root is often calculated, which is slower as both the positive and negative solutions are considered in the predicates. The best case scenario is for `donald`, which exhibits a 25% performance gain over the base version. No direct cause for this unexpected performance gain could be found, the most probable reason is that many measures are omitted in the new version that prevent negative values from occurring.

With the inclusion of the new *Sparse* domain alongside the other extensions, on average the benchmarks suffer a performance loss of 18%. The worst case test is `queens 29` with 44% and the best case, `interval 256`, has a 23% performance gain over the base version. The `queens 29` test creates a lot of holes on a small domain which is more efficient with a bit-vector than *MinMax* chunks. The `interval 256` test often iterates on domains: this is more efficient in the new *Sparse* domain because the n^{th} element can be instantly obtained from a *MinMax* chunk. The original version has to iterate over the bit-vector until the n^{th} element is found.

[5]The results can be reproduced with version 1.4.4 of GNU Prolog for the current version and the git branch `negative-domain` for the new version.

Note that these benchmark tests do not utilize the enhanced capabilities of the new solver. For instance, the new *Sparse* domain does not restrict the domain to smaller bounds. It is therefore difficult to make a fair comparison.

7.2 Overview of the extensions

The new solver is now fully capable of handling negative values in the FD variables. All mathematical predicates have been updated to correctly handle this extension. As an example, consider the constraint below.

```
| ?- fd_domain(X,-10,10), X * 2 #= Y.

X = _#0(-10..10)
Y = _#20(-20..20)
```

The new solver now prevents overflows from occurring, making the solver more user-friendly. As an example, consider the constraint below.

```
| ?- X * Y #= Z.

X = _#3(-268435455..268435455)
Y = _#20(-268435455..268435455)
Z = _#37(-268435455..268435455)
```

The new solver has a redesigned *Sparse* representation that no longer decreases the domain bounds to `[0..fd_vector_max]`: no values are lost anymore with the new design (rendering the *extra_cstr* flag useless). As an example, consider the constraint below.

```
| ?- fd_domain(X,-300,100), X #\= -20.

X = _#0(-300..-21:-19..100)
```

8 Future work

While the results show that the extensions only cause a limited slowdown factor, there is much room for improvements.

The measures taken to prevent overflows can be optimized further. For instance, when the run-time domain bounds imply that no overflows can occur; for instance if X and Y are in `[0..10]` there is no need to check for overflow in the constraint $X \times Y = Z$, since domains are reduced monotonically. As seen in Section 4.2, supporting negative numbers for $X^A = Y$ implies testing the parity of A. At present this is done every time the constraint is reactivated, however, with a slightly more complex compilation scheme, there will be two versions of the execution function (see 2.1.1): one specialized for even As and another for odd. The installation function would be responsible to select the adequate execution function, depending on the actual value of A at run-time. This will entail enriching the FD language to be able to express user-specified installation procedures.

It will definitely be interesting to combine the new *Sparse* domain representation with bit-vectors, whenever applicable. Similarly, instead of using a (doubly-linked) list for maintaining chunks, a tree-structure is likely to be more efficient. Ohnishi et al. [8] describe how a balanced tree structure is realized on interval chunks. Incorporation of this structure should improve the time complexity on insertion and deletion from $O(n)$ to $O(\log n)$ (for n as the number of chunks) in worst case scenarios.

As a future extension it is also interesting to ease the access of GNU Prolog from a user-point of view, by adding support for the C interface with shared libraries to make GNU Prolog accessible in other languages. Also, the inclusion of multiple datatypes, such as floats or date objects, in the FD solver is useful for practical purposes.

9 Conclusion and Discussion

Relating back to the intial research questions, it can be concluded that they have all been answered:

1. GNU Prolog is extended to support negative values in the FD solver, by properly updating the mathematical predicates.
2. Overflows are now prevented by reordering constraints and checking if values exceed variable bounds (and restricting this from happening).
3. The *Sparse* representation has been redesigned with a list of interval chunks, making sparse domains no longer extra constrained.

The extensions to the GNU Prolog FD solver allow it to more gracefully handle real-world problems. Aside from the extensions, the development environment of GNU Prolog has been improved by presenting a couple of test frameworks.

The performance evaluation of the initial (and most likely suboptimal) implementation shows encouraging results: the slowdown is quite acceptable, in the order of 20%. Furthermore, future work includes ways to further reduce the impact of these design options, with the aim to reclaim the lost performance.

The new solver has been preliminary tested by Axini: all features seem to work and no problems were found. The main developers of GNU Prolog plan to include the implemented features in a future build.

References

[1] Krzysztof R. Apt. *Principles of constraint programming.* Cambridge University Press, 2003.

[2] Philippe Codognet and Daniel Diaz. Compiling constraints in clp(fd). *Journal of Logic Programming*, 27(3):185–226, 1996.

[3] Rina Dechter. *Constraint processing.* Elsevier Morgan Kaufmann, 2003.

[4] Daniel Diaz, Salvador Abreu, and Philippe Codognet. On the implementation of GNU Prolog. *TPLP*, 12(1-2):253–282, 2012.

[5] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, 1989.

[6] Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(fd). In Andreas Podelski, editor, *Constraint Programming*, volume 910 of *Lecture Notes in Computer Science*, pages 293–316. Springer, 1994.

[7] Vianney Le Clément de Saint-Marcq, Pierre Schaus, Christine Solnon, and Christophe Lecoutre. Sparse-Sets for Domain Implementation. In *CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*, pages 1–10, September 2013.

[8] Shuji Ohnishi, Hiroaki Tasaka, and Naoyuki Tamura. Efficient representation of discrete sets for constraint programming. In Francesca Rossi, editor, *CP*, volume 2833 of *Lecture Notes in Computer Science*, pages 920–924. Springer, 2003.

[9] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming.* Elsevier, 2006.

[10] David H. D. Warren. An abstract prolog instruction set. Technical Report 309, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Oct 1983.

Appendices

This part contains information about the file structure, control flow and instructions for the development and verification of the GNU Prolog FD solver.

Appendix A Program structure and control flow

This appendix contains information on the compilation scheme, relations between files and the control flow for the FD solver of GNU Prolog.

The compilation scheme for GNU Prolog can be found in Figure 4[6]

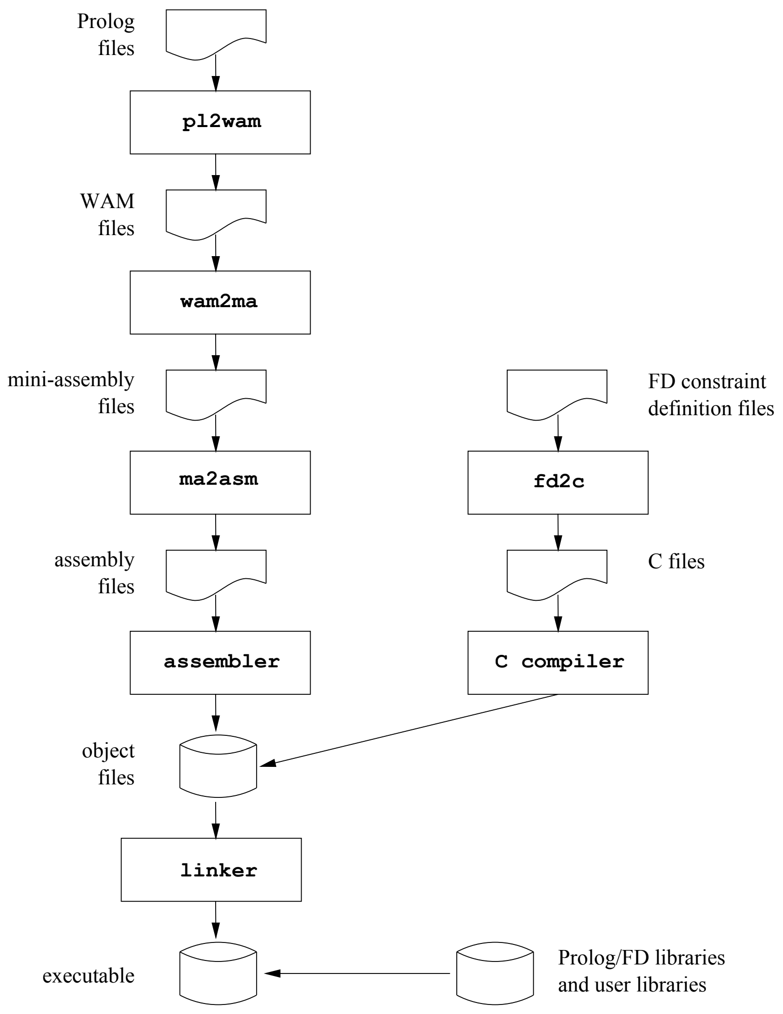


Figure 4: Compilation scheme for GNU Prolog.

A Prolog source file is compiled in several stages to produce an object file that is linked to the GNU Prolog libraries. The Prolog source is first compiled to obtain a Warren Abstract Machine or WAM [10] file. This WAM file is then translated to a language close to a 'universal' assembly language, called mini-assembly (MA). This MA file is then assembled to an object file for the specific target machine. The object file is linked to the GNU Prolog libraries using the FD constraint definition files, which contain the FD predicates. From this, an executable is obtained.

[6]This scheme can also be found in the manual of GNU Prolog: `http://www.gprolog.org/manual/gprolog.html`.

Regarding the file structure of GNU Prolog, only the FD constraint solving scheme is considered. A visual representation of this file structure can be found in Figure 5. Note that this is by no means a complete overview; the purpose of the diagram is to present a basic outline for the relations between files and to briefly indicate the intentions of each file.

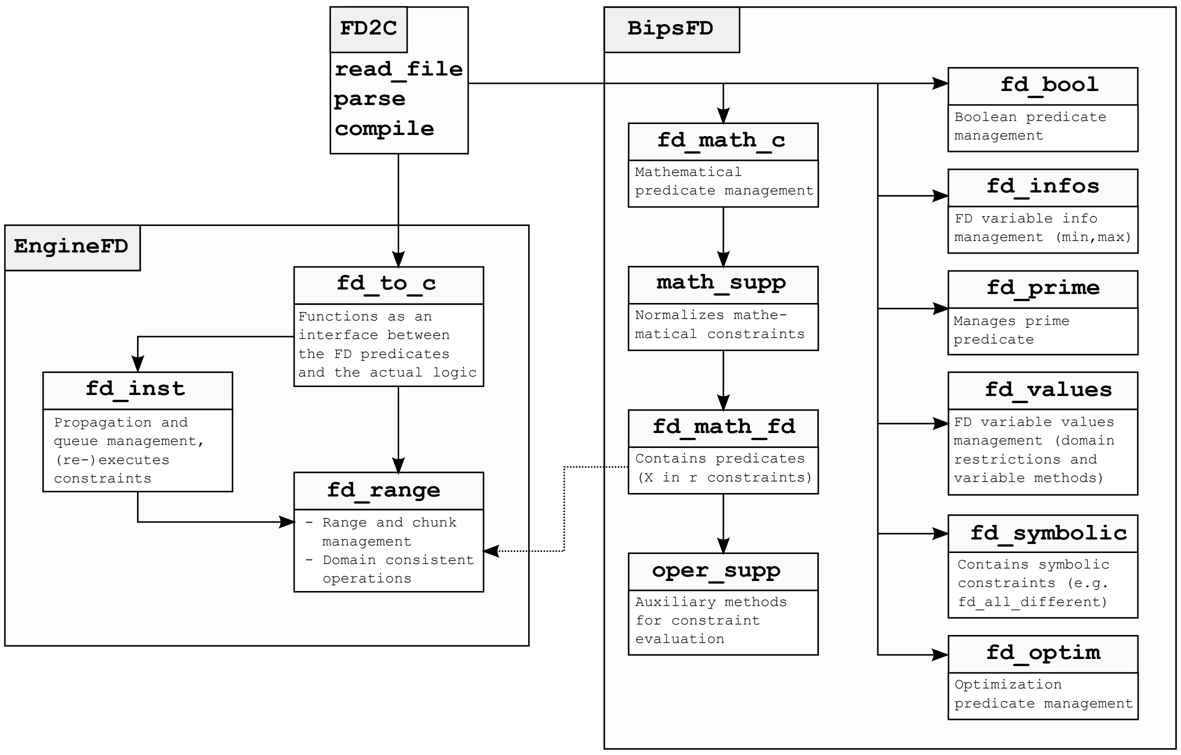

FD2C
read_file
parse
compile
BipsFD
fd_bool
Boolean predicate management
fd_math_c
Mathematical predicate management
fd_infos
FD variable info management (min,max)
EngineFD
fd_to_c
Functions as an interface between the FD predicates and the actual logic
math_supp
Normalizes mathematical constraints
fd_prime
Manages prime predicate
fd_inst
Propagation and queue management, (re-)executes constraints
fd_values
FD variable values management (domain restrictions and variable methods)
fd_math_fd
Contains predicates (X in r constraints)
fd_range
- Range and chunk management
- Domain consistent operations
fd_symbolic
Contains symbolic constraints (e.g. fd_all_different)
oper_supp
Auxiliary methods for constraint evaluation
fd_optim
Optimization predicate management

Figure 5: File structure for the GNU Prolog FD constraint solver.

FD constraints from Prolog source files (or from the interpreter) are parsed in the `FD2C` file structure. These constraints are compiled to eventually apply the FD predicates from the GNU Prolog libraries and to retrieve the resulting FD variables.

Before this is possible, the FD engine (`EngineFD`) instantiates FD variables. The `fd_to_c` file contains macros to connect basic operation calls from the `FD2C` compiler to the actual implementation. For example, `fd_range_interval(r, t_min, t_max)` instantiates the FD variable *r* (in `fd_inst`) with the domain `[t_min..t_max]` and `fd_range_add_value(r, t)` updates *r* by applying a domain-consistent addition with value *t* (in `fd_range`). The file `fd_inst` keeps track of the propagation and queue management, meaning that whenever an FD variable changes the constraints for this variable are re-executed and consequently related FD variables are updated. The range (and chunk in the new version) management is realized in `fd_range`. Domain consistent constraints are implemented here as these closely relate to the *Sparse* design of a range.

The built-in predicates (`BipsFD`) are subdivided in seven types of predicates:
- `fd_math` contains mathematical predicates (e.g. `X * Y #= Z`).
- `fd_bool` contains predicates over boolean formulas (e.g. `X #==> Y`).
- `fd_infos` contains predicates about the minimum and maximum values (e.g. `fd_max_integer(X)`).
- `fd_prime` contains a predicate that restricts FD variables to prime values: `fd_prime(X)`.
- `fd_values` contains predicates for managing values for FD variables (e.g. `fd_domain(X,1,10)`).
- `fd_symbolic` contains symbolic predicates (e.g. `fd_all_different([X,Y])`).
- `fd_optim` contains predicates for optimizing specific cases (e.g. `fd_minimize(X#>5,X)`).

The solver is able to simplify constraints to their basic forms. For mathematical constraints `math_supp` applies normalization techniques. For instance the constraint `A * B #= C * D` is reduced to two constraints of the form `X * Y #= Z`. This means that a finite number of basic predicates is able to represent infinitely many constraints.

Note that the *bound* consistent and *domain* consistent predicates differ from each other as explained in Section 2.1.1. In the implementation, bound consistent mathematical predicates are performed in `fd_math_fd` and domain consistent mathematical predicates are executed in `fd_range`. The file `oper_supp` contains auxiliary methods for the power operation.

Appendix B   Development instructions

This appendix contains instructions specifically aimed for future developers of GNU Prolog, in particular for developers of Axini.

Using GNU Prolog without installation: Directions for installing GNU Prolog are provided in the `INSTALL` file, found in the root directory of GNU Prolog. For development purposes it is often useful to manage different versions. For this, it is possible to 'install' GNU Prolog for a single terminal session:

```
$ cd src
$ . SETVARS
$ ./configure
$ make
```

here, `SETVARS` will update the `PATH` with the files of the current directory. After the `make` call, the user can use GNU Prolog by executing the command: `gprolog`.

Configuration flags: GNU Prolog can be compiled with several optional features. Two interesting flags to be set for future development are `--with-c-flags=debug` and `--disable-regs`. The first flag can be used for significantly speeding up the compilation process as no optimizations will be applied. The second flag will tell the compiler to not use the machine registers: this might be useful for cases where other processes require machine registers for correct functionality.

Debugging GNU Prolog: The file `/src/BipsFD/math_supp.h` contains a `DEBUG` flag that can be set on or off. Initially this flag is turned off since the debug information prints on the standard output. The debug output contains information on normalizations that take place in the expressions (the implementation for this is found in `/src/BipsFD/math_supp.c`) and it shows which predicates are called from the FD files. The latter has been very useful for testing the modifications applied in `/src/BipsFD/fd_math_fd.fd` (for including negative values and overflow prevention). Consider the following example:

```
| ?- X * 5 #>= 4 + 6.

*** Math constraint : 4+6 #=< _23*5
l_nb_monom:0   r_nb_monom:1
normalization: 10 #=< 5*_23

'pl_ax=y'(5,_23,_#0(-268435455..268435455))
'pl_x>=c'(_#0(-268435455..268435455),10)

X = _#20(2..53687091)
```

here, the normalization process reduces $4 + 6$ to 10. While this is a simple example, the normalization is able to transform complex expressions to simple polynomials. As mentioned in Appendix A, these expressions are then split up in smaller parts and thereafter fed to the predicates. Here, two predicates are called: `pl_ax=y` and `pl_x>=c` and the debug information also shows the arguments passed to these predicates.

Removing negative values: As mentioned in Section 4.2 for backwards compatibility, an environment variable, `GP_FD_POSITIVE_ONLY`, is created in `src/EngineFD/fd_range.h`. If this variable is set, the solver will set `min_int` to 0. This is sufficient for reverting to the original domain bounds. Note that the predicates are not modified by this.

Prolog disjunction and choice-points: In case a logical OR is desired over two FD constraints, the Prolog disjunction (`;`) is an effective option[7]. As an example, consider the following constraint:

```
| ?- fd_domain(X,1,100), (X #> 30 ; X #< 10).

X = _#0(31..100) ? ;

X = _#0(1..9)
```

here, a *choice-point* is created after `fd_domain(X,1,100)`, this means that the domain for X is stored on the internal stack. Then, `X #> 30` is taken into account, which reduces X to `[31..100]` and X is printed. If the user requests another solution, the Prolog engine backtracks and restores the domain for X. Finally `X #< 10` is evaluated, resulting in the domain `[1..9]`.

B.1 Testing

Unit testing the sparse domain: The C program `/src/EngineFD/test_fd_range.c` is a unit test suite that is used for verifying the new sparse domain. It tests the methods implemented in the file `/src/EngineFD/range.c`. The test suite is executed with the command: `make check`, presuming that the current directory is `/src/EngineFD`. To give an idea of these test cases, consider the following code snippet:

```
r = create_range("17..25:30..40");
r1 = create_range("3..11:20..40");
Pl_Range_Inter(r,r1);
range_test("Sparse with Sparse - Test 8", r, "20..25:30..40");
```

here, a range (or domain) for an FD variable is created via a string representation (`create_range` parses this and constructs a range). `Pl_Range_Inter` is the method to be tested from `/src/EngineFD/range.c`, this method will store the intersection of the FD variables `r` and `r1` in `r`. The macro `range_test` then checks if `r` is equal to the domain `[20..25:30..40]` (it will also check if all pointers are set up correctly in `r`). If this is not the case, the test will fail and the user is informed of this failure. For demonstrative purposes, a failure of a test case looks as follows (the intersection is omitted in this case):

```
TEST FAIL: Sparse with Sparse - Test 8

Sparse Range is 17..25:30..40 instead of 20..25:30..40

In Pl_Range_Inter
```

here, the test message is shown (this indicates which tests fails). The difference between the actual and expected range is shown thereafter and lastly the method that is currently being tested is outputted. For statistical data, 28 different methods (this is actually more as auxiliary methods are not included) are tested with a total of 419 test cases.

Testing the mathematical predicates: The files `src/testcases.pl` and `src/generated_tests.pl` both consist of a series of test cases specifically aimed at testing the mathematical predicates (from `src/BipsFD/fd_math_fd.fd`). Compilation is done with `gplc [filename]`, which creates an executable with the same name. `src/testcases.pl` executes a number of constraints in the form of mathematical predicates. The results for these tests (the resulting FD variable bounds) are printed on the standard output to be verified by the user (or by comparing the results with a provided correct output file). `src/generated_tests.pl` is similarly designed in that it also outputs the FD variable bounds. The difference is that this program consists of a wide variety of test cases by testing different configurations in the constraint `X * Y #= Z`. The reason for this 'extra' program is because the redesign of said constraint became rater complex. There are however two downsides to these test programs. First, the program fails if an FD variable cannot take any values (and represents the *empty* domain). A consequence is that

[7]The operator `#\/` should not be used for this purpose. While this also defines an OR expression, this is specifically designed for boolean FD variables and assumes that the FD variables are in the domain `[0..1]`.

the program cannot test for false negative errors. Second is that the constraints and expected results are separated, making the verification process tedious.

Automatic testing with the interpreter: The C program `src/test.c` is initially designed to overcome the drawbacks of the Prolog test programs and to serve as a universal testing framework. This test suite is designed similarly as the range unit tests in the sense that one test case at a time is evaluated. The GNU Prolog interpreter is used for evaluating test cases by first writing an input file with the to be tested constraint. Second, the GNU Prolog interpreter is called with this input file and the output is written to an output file. Third, the contents of the output file are parsed and compared with the expected output. A test case looks as follows:

```
test_constr("fd_domain(X,-100,100), X #\\= 5.",  "X = [-100..4:6..100]");
```

here, the input and expected output are close together for easy verification. A test failure results in the following:

```
TEST FAIL: X #> 268435455.

found:
  X = (268435455)
expected:
  no

In test_gt_lt
```

in this case, an overflow preventive measure has been disabled. With the provided information it should be clear for the tester what goes wrong. Observe that this suite is able to test for false negatives unlike the Prolog test instances. Please note that this suite is not yet fitted with many test instances; preliminary work has been done for future developers.

Benchmark tests: the directory `/examples/` contains sets of test programs for GNU Prolog. The subdirectory `/examples/ExamplesFD/` contains specific tests that utilize features of the (original) FD solver. These test instances are used for comparing the performance differences between the new and the original version (executing `./CHECK-FD-BENCH` should provide equivalent results as discussed in Section 7.1). So far there are no example programs that utilize the new features of the FD solver.